

Using Histories to Implement Atomic Objects

Pui Ng

Department of Computer Science

University of Illinois at Urbana-Champaign

1304 W. Springfield Ave

Urbana, IL 61801

November 24, 1987

Abstract

In this paper we describe an approach of implementing atomicity. Atomicity requires that computations appear to be all-or-nothing and executed in a *serialization order*. The approach we describe has three characteristics. First, it utilizes the semantics of an application to improve concurrency. Second, it reduces the complexity of application-dependent synchronization code by analyzing the process of writing it. In fact, the process can be automated with logic programming. Third, our approach hides the protocol used to arrive at a serialization order from the applications. As a result, different protocols can be used without affecting the applications. Our approach uses a history tree abstraction. The history tree captures the ordering relationship among concurrent computations. By determining what types of computations exist in the history tree and their parameters, a computation can determine whether it can proceed.

1 Introduction and Motivation

Atomicity [Gra78, Lam80, LS83, S*85, W*85] has been accepted as a powerful and intuitive concept to control the complexity of concurrency and recovery in a distributed system. In a system supporting atomicity, computations are executed as *atomic transactions*, which are *failure atomic* and *serialized*. Failure atomicity means that the *outcome* of a transaction is either *committed* or *aborted*: either all of a transaction is executed or it appears that none has been executed. "Serialized transactions" means that transactions appear to execute serially in a global *serialization order*, even though

they may be overlapped in actual execution. A concurrency control algorithm is needed to ensure atomicity.

Many concurrency control algorithms have been proposed in the literature. Common examples include the two-phase locking and timestamp ordering algorithms[Gra78,BG81]. Many systems that employ these concurrency control algorithms have been successfully implemented[IMS84,W*82,A*76,Lis85]. Using these systems is relatively simple: a programmer is only required to specify the boundaries of an atomic transaction. Concurrency control is performed transparently by the system. As a result, it is also easy to substitute one concurrency control algorithm for another without affecting the application programs.

In many concurrency control algorithms, a system is modeled as a collection of objects on which read and write operations are performed. The model imposes a limit to concurrency because, to guarantee failure atomicity, an object written by an incomplete transaction cannot have its new value read by other transactions until the incomplete transaction is committed. Similarly, to guarantee serializability, for every transaction T intending to write a new value, the system must ensure that the new value is read only by other transactions that are serialized after T . To increase concurrency, many researchers have suggested utilizing the semantics of an application[Gar83,Wei84,SS84]. For example, two transactions that increment some kind of counter object can proceed concurrently because increments are commutative. Moreover, an increment operation really does not read the value of the counter at a logical level, although at a lower level a read operation may be needed. A transaction should be able to increment the counter object even when the previous transaction that increments the counter is still uncommitted. The need for a higher level of concurrency becomes more pronounced in systems where *long* atomic transactions are executed[Ng86] and systems with localized concurrency bottlenecks.

Unfortunately, introducing application semantics has two drawbacks. First, it makes writing applications with atomic transactions more complicated. For example, to capture application semantics, some researchers have extended read/write locks to logical locks[SS84,Wei84]. "Increment locks" that does not conflict with one another can be used in the example above when an increment operation is invoked. In addition to specifying these logical locks and pairs of logical locks that conflict, the programmer must also provide some form of logical recovery so that the counter object may be in the correct state under all possible combinations of outcomes of the increment transactions.

Second, it is not clear how an application program can be written without exposing the under-

lying concurrency control algorithm. An essential component of a concurrency control algorithm is its *serialization protocol*, which determines the serialization order. For example, the serialization order is determined by the order of commitment in two-phase locking. In approaches using logical locks, the application programmer is aware of the underlying two-phase locking algorithm and hence its serialization protocol. Changing the serialization protocol is not straightforward and would require at least changing the concurrency control algorithm and rewriting the application programs. Perhaps even worse, it requires familiarizing the application programmers with a new algorithm.

The goal of this paper is to deal with these two problems using a *history* abstraction as the basis for synchronization and recovery. We will describe how an application programmer can write synchronization and recovery code based on the semantics of the application. We will argue that this process is not an ad hoc one but, rather, follows a structured pattern. Consequently, even though each application may have different semantics, writing application-specific synchronization and recovery code and arguing about its correctness are simplified. We will also show that our approach provides a basis for automating application-specific synchronization and recovery. Finally, we will illustrate how the history abstraction can be incorporated with the serialization protocols of a large class of concurrency control algorithms. Changing the serialization protocol in the internal implementation of the history abstraction will not affect the application-specific synchronization and recovery code written using the abstraction.

The rest of this paper is organized as follows. In section 2 we will describe our system model. Since application semantics is needed in our solution, we will also describe how it can be specified using informal specifications. Section 3 gives an overview of our paradigm for synchronization. Section 4 describes the interface of the abstractions provided to support our synchronization paradigm. Section 5 is a simple example illustrating our paradigm. Section 6 improves on the example in section 5 by providing more concurrency. Section 7 discusses how to deal with operations that cannot proceed immediately. Finally, section 8 describes how application-dependent synchronization and recovery can be automated.

2 System Model and Specification of Semantics

A system is modeled as a collection of *atomic objects* [Wei84]. Each atomic object supports a set of operations. A transaction consists of a sequence of operations invoked at different atomic objects.

Atomic objects cooperate to make transactions atomic. The last operation invoked by a transaction on any object is either a *commit* operation or an *abort* operation, which are supported by every atomic object so that it can be informed of the outcome of the transaction. As part of concurrency control, atomic objects follow a serialization protocol, which determines a global serialization order of the transactions. For example, in the two-phase locking algorithm, the serialization order is determined by the order in which transactions are committed. In most timestamp ordering algorithms, the serialization order is determined by a timestamp acquired by each transaction at its beginning. In some optimistic concurrency control algorithms [Dav84], the serialization order is determined by the order of transactions in a dependency graph. The atomicity of transactions is ensured by having *each* atomic object maintain the appearance that operations are executed in the global serialization order determined by the serialization protocol.

Similar to the approach used in the paper by Liskov and Weihl [LW86], we will specify the semantics of an atomic object informally using a state machine. An atomic object is specified with four components: a set of possible *states*, an *initial state*, a set of possible *transitions*, and *rules* that determine how the states of the atomic object are changed by the transitions. A transition corresponds to an operation invoked at the atomic object and the result returned.¹ For example, the specification of a bank account object may look like the following:

Possible states: *non-negative real numbers*

Initial state: 0

Possible transitions: *deposit_x_okay*, *withdraw_x_okay*, *withdraw_x_insuf*, *read_balance_x*

Rules for state changes: $N(s, \text{deposit_}x_okay) = s + x$

$$N(s, \text{withdraw_}x_okay) = s - x \text{ if } s \geq x$$

$$N(s, \text{withdraw_}x_insuf) = s \text{ if } s < x$$

$$N(s, \text{read_balance_}x) = s \text{ if } s = x$$

where s is the current state of the state machine, x is any non-negative real number, and N is a function defining how state changes in the state machine with each incoming transition. Notice that N is actually a partial function, since for some pairs of state s and transition t , $N(s, t)$ is undefined.

The bank account object supports three operations: *deposit*, *withdraw*, and *read_balance*. The *deposit* operation takes one argument of type *real*, which is represented by the symbol x , and

¹The specification has incorporated the result returned by an operation as part of the transition, rather than as an output caused by the transition. This is merely a notational convenience.

adds that amount to the balance. It always returns *okay*, which means that the operation is successfully performed. The *withdraw* operation also takes one argument (x). It may return *okay* or *insuf* depending on whether there is sufficient funds in the balance to cover the withdrawal. The *read_balance* operation returns the current balance, which is represented by the symbol x .

When there is no concurrency in the system (i.e., transactions are executed serially), an implementation of an object meets the specification as long as *any transition sequence that it generates in response to invoked operations is valid*, and a transition sequence is *valid* if it causes the state machine to go from the initial state to one of the possible states. For example, in the bank account example above, the sequence $\{deposit_30_okay, withdraw_20_insuf\}$ is *invalid* because the first transition causes the state to change from 0 to 30, and $N(30, withdraw_20_insuf)$ is undefined. The specification models our expectation that withdrawing \$20 should not return with insufficient funds after we have deposited \$30.

When there is concurrency in the system, instead of considering the actual transition sequence generated by an atomic object, we should consider an hypothetical sequence that consists only of committed transitions and is ordered according to the global serialization order. We call this hypothetical sequence the *serialized sequence*. An implementation of an atomic object is considered to meet the specification if *any serialized sequence it generates is valid*. For example, suppose $\{deposit_10_okay, withdraw_40_insuf, deposit_50_okay\}$ is the actual transition sequence generated. Furthermore, suppose the transaction that executed the first *deposit* transition is eventually aborted, and the transaction that executed the second *deposit* transition is serialized before the transaction that executed the *withdraw* transition. The serialized sequence would be $\{deposit_50_okay, withdraw_40_insuf\}$. As it is *invalid*, the implementation does not meet the specification. Since aborted transitions are left out in all atomic objects, a correct implementation appears to be failure atomic. Since all atomic objects follow the same serialization order, transactions appear to execute serially in that order.

3 Overview of Synchronization

In many systems that support atomic transactions, synchronization is transparent[W*82,Lis85]. An account object like the one described above would typically be implemented with a real number. A *deposit* or *withdraw* operation would be implemented with a read and a write operation on the real number. Synchronization would be performed transparently during the read and write operations.

For example, a read lock and a write lock associated with the real number can be acquired.

Synchronization is explicit in our approach. The state of an atomic object is represented as a history of previously executed transitions. The execution of an operation on the atomic object is implemented as an *addition* to this collection of transitions. Explicit synchronization is needed before the addition to determine whether the current operation can proceed immediately or has to be retried. The following is a simplified outline of the code for a typical operation of an atomic object.

```
var state = <history of transitions>
operation1 = procedure(<argument declarations>) return(<result declarations>)
    if <synchronization code to determine whether operation can proceed>
        then <insert transition for operation1 into history of transitions>
        else <retry later>
    end procedure
```

In this section we will describe how the application-dependent synchronization code can be written. The goal of the synchronization is to guarantee that the serialized sequence generated is valid, based on the information kept in the history of previously invoked transitions.

Given this goal, the synchronization problem becomes trivial if each atomic object has complete knowledge of the serialization order and transaction outcomes. In other words, the atomic object has complete knowledge of the serialized sequence. When a new operation is invoked, an implementation for an atomic object would simply return a result such that the resultant serialized sequence is valid. For example, if an account object implementation has complete knowledge of the serialized sequence, a *read_balance* operation would never be delayed as the balance can be determined by crediting/debiting the deposits/withdrawals in the order of the serialized sequence.

Unfortunately, this is not the case as incomplete transactions can either commit or abort, and some serialization protocols, such as the one used in two-phase locking, do not determine the serialization ordering between two transactions until they commit. In other words, there may be different possible values for the serialized sequence. The atomic object may not know which one is the right value. It is this lack of knowledge that causes loss of concurrency. For example, a *read_balance* operation cannot proceed if it is uncertain whether a previous update transition will be ordered before itself in the serialized sequence. Fortunately, in many situations, the semantics of an application would allow an implementation to return a result even though there may be

many possible serialized sequences. For such a result to be returned, *each* of the *possible* resultant serialized sequences must be valid.

To illustrate with an example, consider an account object that had processed a committed *deposit* operation that deposited \$100 and a *withdraw* operation that withdrew \$40. The transaction that invoked the *withdraw* operation is incomplete. Suppose another *withdraw* operation that tries to withdraw \$50 is invoked. Furthermore, suppose the account object knows that the *deposit* operation is serialized before the two *withdraw* operations but the serialization ordering between the two *withdraw* operations is unknown. Normally, if we use two-phase locking with read/write locks, the second *withdraw* operation will be delayed because both *withdraw* operations have to acquire a write lock. However, since in all the possible serialized sequences, there is always enough money to cover the second withdrawal, an *okay* response can be returned immediately. (In other words, all the possible serialized sequences that include a *deposit_50_okay* transition will be valid.) Similarly, if the second *withdraw* operation tries to withdraw more than \$100, an *insuf* response can be returned immediately. Notice that if the second *withdraw* operation tries to withdraw, say, \$70, it will be delayed. This is because whether the withdrawal will succeed depends on whether the first *withdraw* operation will commit. In other words, regardless of whether a *withdraw_70_okay* transition or a *withdraw_70_insuf* transition is added, some of the possible resultant serialized sequences are invalid. Consequently, neither the *okay* response nor the *insuf* response can be returned immediately.

From our example above, we see that the required synchronization involves finding a result to a newly invoked operation such that *all* possible resultant serialized sequences are valid. In order to analyze this process, we will classify the possible transitions of an atomic object into *observer transitions* and *mutator transitions*. An observer transition allows information to be derived about the state of the object (i.e., at least two different results can be returned to this transition's operation depending on the state of the object). For example, the *read_balance* and *withdraw* transitions are observer transitions (the latter allows the caller to determine whether the balance is more than the amount to be withdrawn). A mutator transition t changes the state of the object (i.e., $N(s, t) = s'$ s.t. $s \neq s'$). For example, the transitions *deposit_x_okay* and *withdraw_x_okay* are mutator transitions. A transition can be both a mutator and an observer. An observer transition t_i has a *valid* observation with respect to a sequence $t_1.t_2...t_i...t_n$ of transitions if $N(...N(N(I, t_1), t_2), ..., t_i) \neq \perp$, where I is the initial state and N is the state transition function. In order for a sequence Q to be valid, each of the observer transitions in Q must have a valid observation with respect to Q .

Two synchronization requirements have to be met before a result can be returned to an operation (i.e., a new transition can be added). The two requirements apply when the new transition is an observer and a mutator respectively. In the first requirement, the new observer transition must have a valid observation with respect to any possible serialized sequence. In the second requirement, for each possible serialized sequence Q , each observer transition serialized after the new mutator transition must have a valid observation with respect to Q . For example, in order to return *okay* to a *withdraw* operation (*withdraw_x_okay* is both an observer and a mutator), we must make sure that there is enough money to cover this withdrawal (the observation of this transition is valid) and there is still enough money to cover other withdrawals serialized after this one (the observation of other transitions serialized after this one remains valid). Synchronization can be viewed as enumerating all the possible serialized sequences and testing the two requirements.

Initially, it may seem inefficient to have to enumerate all the possible serialized sequences. Fortunately, optimizations are possible. Instead of enumerating all the possible sequences, testing for *patterns of transitions* can identify invalid observations. For example, the observation of a *withdraw_x_okay* transition may be invalid if there are uncommitted *deposit/withdraw* transitions that may be serialized before it. The presence of the uncommitted transactions indicates the possibility of at least two different serialized sequences, each having a different balance. Notice that we can ignore non-mutator transitions such as *read_balance* and *withdraw_y_insuf* transitions. Their position relative to the *withdraw_x_okay* transition in a sequence does not affect the validity of the latter's observation.

A more detailed analysis of transition patterns can sometimes provide more concurrency, and still avoid enumerating all the possible serialized sequences. In the previous example, we can consider a possible "worst-case" sequence. It happens when all the *deposit.y_okay* transitions that may be aborted or serialized after the new *withdraw* transition eventually do, and all the other *withdraw_x_okay* transitions that may be committed and serialized before this one also eventually do. If the observation of this *withdraw* transition is valid with respect to this worst-case sequence, then the observation will be valid for all other possible serialized sequences. This is because the observed balance in all other sequences will be more than or equal to that in the worst-case sequence.

Summarizing, writing application-dependent synchronization code follows a structured pattern. First, transitions should be classified as observers and mutators. Second, before adding an observer transition, we should determine which mutator transitions may affect the validity of its observation. Code should be written to determine whether they exist and may be serialized before the

new transition. Third, before adding a mutator transition, we should determine which observer transitions may have their observation invalidated by the new transition. Code should be written to determine whether they exist and may be serialized after the new transition.

4 History Trees

To support the kind of synchronization described above, we will describe a *history* abstraction. It can be used by an atomic object implementation to capture the history of previously processed transitions, their serialization ordering, and the outcomes of the corresponding transactions, which in turn determines the set of possible serialized sequences. An instance in the history abstraction can be visualized as a *tree*. In this tree, each node is a *transition record* that captures the information in a transition. An edge exists between two transition records if their serialization order is known and is in the direction of the edge.

In a simple implementation, an atomic object is expected to be associated with a single history tree. A transition record will be inserted into the history tree each time an operation is processed by the atomic object. Transition records are also periodically deleted from the history tree after they are committed and their serialization order determined.

A transition record contains the name of the operation, the values of the argument(s) and result, and the status of the transaction that invokes the operation (which is either committed, aborted, or unknown). To simplify our examples in sections 5 and 6, we will not show how these fields are filled in. The reader can assume that when an operation is invoked, a transition record is created and most of the fields are initialized. More updating takes place when the operation returns and when an atomic object is informed of the outcome of a transaction. Primitives are provided to read the fields in a transition record. For example, *t.arg1* returns the first argument of the transition *t*. We also assume that a distinguished variable *this_transition* is associated with the current transition record when an operation is processed.

In addition to history trees and transition records, we also make use of *template records*. Their purpose is for pattern matching with transition records. For example, a template like *committed_withdraw_x_okay* matches any successful *withdraw* transitions that have committed. The following is a list of the operations provided by a history tree *h*. We assume a syntax of "*h.operation(arguments)*" in invoking these operations, where *h* is the identifier of a history tree.

`new = procedure()`

% initialize h to an empty history tree.

delete_first = procedure() returns(transition)

% deletes and returns a committed transition that is serialized before all other
% transitions in h. If such a transition does not exist yet, the invoking process
% will be suspended until it does. Meanwhile, any monitor locks acquired by the
% suspended process are released. Those monitor locks will be acquired again
% before resuming the suspended process.

match = procedure(t: template) returns(array[transition])

% returns a sequence of the transitions in h that matches t.

exists = procedure(t: template, p: proctype(transition) returns(bool))
returns(bool)

% returns true if there is a transition s in h such that s matches the template t
% and p(s) returns true. Otherwise false is returned. p is an optional argument.
% If p is omitted, only the template t is used to filter transitions in h.

insert = procedure(t: transition)

% inserts t into h.

restrict = procedure(t: transition, after, definite: bool) returns(history)

% returns a sub-history of h, which is also a history tree, but it contains only a
% subset of the transitions in h. The transitions included in the subset depends
% on the two boolean arguments. There are four combinations. If both after and
% definite are true, then all the transitions that are DEFINITELY serialized AFTER
% t are included. A transition t' is definitely serialized after t if there is a
% path of edges leading from t to t' in h and t' is committed. If both after and
% definite are false, then all the transitions that are POTENTIALLY serialized
% BEFORE t are included. A transition t' is potentially serialized before t if
% there is not a path of edges leading from t to t' in h. The other two
% combinations are defined similarly.

The operation *restrict* deserves a special note. It returns a history tree that contains a subset of the transition records in *h*. In addition to the boolean arguments, the exact membership of the history tree returned depends on the current knowledge of the serialization order and transaction outcomes encoded in *h*. For example, with a timestamp ordering serialization protocol in which timestamps are assigned at the beginning of a transaction, *restrict* can be implemented easily based on timestamp comparisons and the status of the transition records. On the other hand, if the serialization order is determined by the order of commitment as in two-phase locking, the exact serialization order can be determined only among committed transitions. The only information we know concerning an uncommitted transition *t* is that it is serialized after any committed transition that has been committed before *t* is invoked (which guarantees that *t* will have a later commitment timestamp). In other words, if the argument *t* is uncommitted, *restrict(t, true, true)* would return an empty history, since we do not know which other transitions will *definitely* be committed and serialized after *t*. *Restrict(t, true, false)*, however, will return a history containing transitions that *may* have later commitment timestamps, which include all uncommitted transitions, and committed transitions that are committed after *t* is invoked.

The discussion above illustrates how the history abstraction hides the internal details of a serialization protocol. Our model of each atomic object possessing partial knowledge of the serialization order and transaction outcomes is sufficiently general to encompass a large class of serialization protocols. [A*83, B*82, BG81, BG83, Dav84, Gra78] Consequently, these serialization protocols can be used within the history abstraction.

5 A Simple Example

In this section we will present a simple example that follows the synchronization paradigm described in section 3. It implements the account object. It is not as concurrent as allowed by the semantics. A more complicated version that provides the extra concurrency will be described in section 6.

```
account = monitor exports read_balance, deposit, withdraw
```

```
% constants declared as abbreviations for transition templates
```

```
read = read_balance_x;
```

```
deposit = deposit_x_okay;
```

```
withdraw = withdraw_x_okay;
```

```

successful_update = deposit or withdraw; % matches any deposit or withdraw
insuf = withdraw_x_insuf;                % transitions with an okay result

AFTER = true; BEFORE = false;
DEFINITE = true; POTENTIAL = false;

var snapshot: real := 0;
    h: history; h.new(); % h initialized to empty history tree

while true do clean_up() end % background process

clean_up = entry procedure() % the monitor lock is acquired
    t: transition := h.delete_first()
    if t.match(deposit) then snapshot := snapshot + t.arg1
        elseif t.match(withdraw) then snapshot := snapshot - t.arg1
        end
    end clean_up % the monitor lock is released

abort = entry procedure(t: transition)
    h.delete(t)
    end abort

deposit = entry procedure(x: real)
    if % make sure no transitions potentially serialized after this transition
        % may have their observation invalidated.
        ~h.restrict(this_transition, AFTER, POTENTIAL).exists(read) and
        ~h.restrict(this_transition, AFTER, POTENTIAL).exists(insuf)
        then h.insert(this_transition); return; end
    retry
end deposit

```

end account

In our example, the account object is programmed as a module with shared data (*snapshot*, *h*) accessed by concurrent processes. To synchronize access to these shared data structures, the module is programmed as a monitor. Exclusive access is provided to a process for the duration of an entry procedure. We assume that the application would structure procedures in such a way that the monitor lock would not be held for an excessive long period of time.

Ignoring the *clean_up* and *abort* procedures for now, we see that the only task performed by the *deposit* procedure is to determine whether any transitions that may be serialized after this (mutator) transition may have their observation invalidated by the deposit. There are two kinds of such transitions. The first kind is *read_balance* transitions. The second kind is *withdraw_x_insuf* transitions. If either kind of transitions exists, the operation has to be retried at some later time. We will explain how the *retry* primitive can be implemented later. If neither kinds of transitions exist, the current *deposit* transition can be inserted to the history tree. Since *deposit* transitions are not observers, we do not have to worry about the validity of any observation of the new *deposit* transition.

In the background, there is a looping process that reduces the size of the history tree by deleting transition records from it. The transition records will be deleted in the global serialization order and only after they are committed. The effects of the deleted transitions are reflected in *snapshot*. Notice that because transitions are deleted in the global serialization order, the value of *snapshot* reflects the balance resultant from the deleted transitions executed in the global serialization order. In the example above, the variable *snapshot* is not accessed by the *deposit* operation at all. Later we will see that it is needed in other operations such as *withdraw* and *read_balance*.

Occasionally, the background process will be blocked because either it cannot yet determine which transition is serialized before all other transitions or that transition has not committed yet. In that case, we have to wait for the serialization order to be determined or some transition to commit. The background process will be suspended but the monitor lock will be released so that other processes can access the shared data.

When a transaction is aborted, the *abort* procedure will be invoked with the transition(s) of that transaction. Recovery is achieved by simply deleting the transition from the history tree.

6 A More Concurrent Deposit Operation

The implementation for the *deposit* operation above is not as concurrent as it could have been. On some occasions, a deposit operation should be able proceed even though there are *withdraw_x_insuf* transitions serialized after itself. This can happen if the amount deposited is so small that it would not have made a difference to the *withdraw* transition. The following more complicated version of *deposit* provides this extra concurrency.

```
account = monitor is read_balance, deposit, withdraw
```

```
....
```

```
deposit = entry procedure(x: real)
```

```
  if % make sure no transitions potentially serialized after this transition
```

```
    % may have their observation invalidated.
```

```
    ~h.restrict(this_transition, AFTER, POTENTIAL).exists(read) and
```

```
    ~h.restrict(this_transition, AFTER, POTENTIAL).exists(insuf,
```

```
                                bind(short_by_less_than, x))
```

```
  then h.insert(this_transition); return
```

```
  end
```

```
  retry
```

```
end deposit
```

```
bind = procedure(p: proctype(real, transition) returns(bool), x: real)
```

```
  returns(proctype(transition) returns(bool))
```

```
  q = procedure(t: transition) returns(bool)
```

```
    return(p(x, t))
```

```
  end q
```

```
  return(q)
```

```
end bind
```

```
short_by_less_than = procedure(x: real, t: transition) returns(bool)
```

```
  return(t.arg1 - highest_possible_balance_observed_by(t) <= x)
```

```
end short_by_less_than
```

```

highest_possible_balance_observed_by = procedure(t: transition) returns(real)
    return(snapshot - accumulate(DEFINITE, withdraw, t)
        + accumulate(POTENTIAL, deposit, t))
end highest_possible_balance_observed_by

accumulate = procedure(certainty: bool, opname: template, t: transition)
    returns(real)

    value: real := 0
    for each s: transition in h.restrict(t, BEFORE, certainty).match(opname) do
        value := value + s.arg1
    end
    return(value)
end accumulate

....
end account

```

The only difference between the *deposit* procedure in this example with the previous one is that we have added a filter *bind(short_by_less_than, x)*. The filter is a procedure (more accurately, a closure) that filters out any *withdraw.y.insuf* transitions that are short by more than *x* dollars under any possible situation. Their existence should not prevent the current *deposit* operation from proceeding because the new deposit would not make a difference to them anyway. In other words, their observation that the balance is less than what they were trying to withdraw is not invalidated by the current *deposit* operation.

The filter is implemented with the *bind* procedure, which is merely a linguistic mechanism to create a closure. It binds *x* to the first argument of the procedure *short_by_less_than*. Inside the *short_by_less_than* procedure, it tests whether a *withdraw.y.insuf* transition (*t*) may be short by less than or equal to *x*. To determine the answer, it finds out the highest possible balance the account may have immediately before executing the withdrawal (*t*) in all the possible serialized sequences. This is the worst-case sequence we mentioned in section 3. If that highest possible balance, added with the current deposit, is more than or equal to the amount to be withdrawn (*t.arg1*), then the withdrawal may be short by less than or equal to *x*.

To determine the highest possible balance, we take the value of the snapshot, which reflects a subset of the update transitions serialized before the withdrawal (those already deleted from the history tree). Then all the deposit transitions in the history tree that *may* commit and *may* be serialized before the withdrawal have to be added in. Finally, all the *withdraw_x_okay* transitions in the history tree that are *definitely* committed and serialized before the *withdraw_y_insuf* transition can be deducted to give a tighter bound.

The following is a possible implementation of the *withdraw* operation which is very similar to the *deposit* operation above.

```

withdraw = entry procedure(x: real) signals(insuf)
  if highest_possible_balance_observed_by(this_transition) < x
    then h.insert(this_transition); signal insuf
    end

  if ~h.restrict(this_transition, POTENTIAL, AFTER).exists(read) and
    ~h.restrict(this_transition, POTENTIAL, AFTER).exists(withdraw,
      bind(exceed_by_less_than, x)) and
    lowest_possible_balance_observed_by(this_transition) >= x
    then h.insert(this_transition); return
    end

  retry
end withdraw

lowest_possible_balance_observed_by = procedure(t: transition) returns(real)
  return(snapshot - accumulate(POTENTIAL, withdraw, t)
    + accumulate(DEFINITE, deposit, t))
end lowest_possible_balance_observed_by

exceed_by_less_than = procedure(x: real, t: transition) returns(bool)
  return(lowest_possible_balance_observed_by(t) - t.arg1 < x)
end exceed_by_less_than

```


7 Retrying an Operation

An operation may not be able to proceed because there is the possibility that its result is invalid or it may invalidate the observation of previous transitions. In those situations, it should be retried. This is the purpose of the *retry* statements in the program. Ideally, we would have a boolean expression as an argument to the *retry* statement indicating the precise condition under which the operation would be able to proceed. The underlying system implementation would then determine when the boolean expression may be satisfied and retry the operation at that time. Unfortunately, this is not feasible for several reasons.

First, with an arbitrary boolean expression, the system implementation has to evaluate it every time the state of the object might have been modified to make the boolean expression valid. In our example, it would involve testing the boolean expression each time the monitor is exited. The cost of evaluating these boolean expressions can be substantial if there are many “false alarms.” Second, for some serialization protocols, delaying is not always an appropriate solution. For example, in a timestamp ordering protocol, the serialization order of a set of transactions is fixed by a unique timestamp that each transaction acquires when it begins. Suppose a transaction that performs mutation (e.g., *deposit*) is invoked later than another transaction that performs observation (e.g., *read_balance*). Furthermore, suppose the first transaction has a smaller timestamp. Because the observing transaction has already returned without observing the effects of the mutating transaction, the mutating transaction cannot be serialized before the observing transaction. Aborting one of the transactions and then restarting it with a later timestamp is the only alternative. On the other hand, delaying is the preferable action if the observing transaction (with a larger timestamp) is invoked after an uncommitted mutating transaction. Given an arbitrary boolean expression, it can be a very difficult, if not impossible, task to require the underlying system implementation to choose the appropriate action.

Some of the solutions that have been suggested in concurrent programming literature[AS83] use condition variables. Each operation will explicitly *wait* and *signal* appropriate condition variables. However, condition variables can only be used to delay, but not restart, operations. Consequently, the use of conditional variables would require additional mechanisms that might betray the underlying serialization protocol.

As a compromise, we require the programmer to specify a boolean expression along with each *retry* statement (not shown in the examples), with the constraint that the boolean expression is

composed only with history tree operations and built-in boolean operators. The boolean expression should indicate the condition under which the operation can proceed. Due to the constraint on the structure of the boolean expression, it may be a sufficient but not necessary condition. The constraint allows the system implementation to determine a set of history tree or transition record operations that may cause the boolean expression to become valid. Given such a set of events, the system implementation can determine the actions that should be taken when an operation cannot proceed immediately, based on the system's perception of the likelihood of the events. For example, suppose the condition for a *deposit* operation to proceed is:

```

~h.restrict(this_transition, AFTER, POTENTIAL).exists(read) and
~h.restrict(this_transition, AFTER, POTENTIAL).exists(insuf)

```

and the serialization protocol is a timestamp ordering algorithm. Given that the *deposit* operation is unable to proceed because some other *read_balance* or *withdraw_x_insuf* transitions are serialized after it, the only event that can cause this condition to become satisfied is to for those transitions to be aborted and deleted from the history tree. Assuming that most transactions eventually commit, delaying is probably less appropriate than restarting the *deposit* transaction with a new (later) timestamp. A more detailed description of the algorithm to implement the *retry* statement can be found in [Ng86]. Summarizing, a constrained boolean expression provided by the programmer seems to both offer efficiency in rescheduling and avoid additional mechanisms that may betray the underlying serialization protocol.

8 Automated Synchronization and Recovery

Automating the application-dependent synchronization and recovery described above is not difficult. A simple approach is to express when an invocation can proceed through the use of logic programming rules. As we will see in an example below, the application-dependent part of synchronization can be captured in a few rules. More importantly, these application-dependent rules can be generated from the specification of an atomic object easily. As a result, automation of synchronization would only require inputting the specification in a suitable format.

Suppose *list_of_sequence* is a list of all the possible serialized sequences. In our notation below, we will assume that a list/sequence $e_1, e_2, e_3, \dots$ can be represented as $e_1.Tail$ where $Tail = e_2, e_3, \dots$. Whether the invoked operation can proceed immediately can be determined by the following rules written in Prolog:

- (1) :- `proceedable(initial_state, list_of_sequence)`
- (2) `proceedable(S, nil)`
- (3) `proceedable(S, Seq.List_of_Seq) :- valid(S, Seq), proceedable(S, List_of_Seq)`
- (4) `valid(S, nil)`
- (5) `valid(S, T.Seq) :- merge(S, T, New_S), valid(New_S, Seq)`

Rule (1) represents the goal that we are trying to prove: whether the invoked operation can proceed given the initial state of the state machine and the list of possible serialized sequences. The proving of the goal can be made more efficient by truncating the common prefix from each of the sequences in *list_of_sequence*, and replacing *initial_state* with the state that the state machine is at after processing the transitions in the common prefix. Rules (2) and (3) state that an invoked operation can proceed immediately as long as each possible serialized sequence in the list is *valid*. Rules (4) and (5) state that a sequence is *valid* as long as each transition in the sequence can *merge* successfully with the snapshot to produce a new snapshot. The rules for *merge* will be application-dependent and determined by the specification of the atomic object. For an atomic bank account object, the rules for *merge* are the following:

- (6) `merge(S, (deposit.X.okay), New_S) :- New_S is S+X`
- (7) `merge(S, (withdraw.X.okay), New_S) :- New_S is S-X, S >= X`
- (8) `merge(S, (withdraw.X.insuf), New_S) :- New_S is S, S < X`
- (9) `merge(S, (read.balance.X), New_S) :- New_S is S, S = X`

The rules above are very similar to the specification we used in section 2. Given the specification in a suitable format, it is not difficult to convert it automatically to rules that can be interpreted by a Prolog interpreter.

9 Conclusion

In this paper we have described an approach of implementing atomicity that has three characteristics. First, it utilizes the semantics of an application to improve concurrency. Second, it helps reduce the complexity of application-dependent synchronization code by analyzing the process of writing it. In fact, the process can be automated with logic programming. Third, our approach hides the actual serialization protocol used to arrive at a global serialization order. As a result, different serialization protocols can be used without affecting the applications.

Using our approach, each application will be able to control its own concurrency by designing its semantics appropriately. With better facilities to harness the concurrency inherent in the semantics of an application, it becomes more feasible to construct large systems with many cooperating applications.

References

- [A*76] M. M. Astrahan et al. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):431-466, June 1976.
- [A*83] D. Agrawal et al. *Multi-Version Optimistic Concurrency Control for Relational Databases*. Technical Report, Department of Computer Science, SUNY at Stony Brook, 1983.
- [AS83] G. R. Andrews and B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3-44, March 1983.
- [B*82] R. Bayer et al. *Dynamic Timestamp Allocation for Transactions in Distributed Systems*, pages 9-20. North-Holland, 1982.
- [BG81] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, 13(2):185-221, June 1981.
- [BG83] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465-483, December 1983.
- [Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456-481, September 1984.
- [Gar83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186-213, June 1983.
- [Gra78] J. N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science Vol. 60*, pages 393-481, Springer-Verlag, 1978.
- [IMS84] *IMS/VS Version 1 System Administration Guide*. 1984.
- [Lam80] B. Lampson. Atomic transactions. In *Distributed Systems: Architecture and Implementation, Lecture Notes in Computer Science Vol. 100*, chapter 11, Springer-Verlag, 1980.

- [Lis85] B. H. Liskov. The Argus language and system. In *Distributed Systems: Methods and Tools for Specification; An Advanced Course, Lecture Notes in Computer Science Vol. 190*, pages 343–430, Springer-Verlag, Berlin, 1985.
- [LS83] B. H. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(7):381–404, July 1983.
- [LW86] B. H. Liskov and W. Weihl. Specifications of distributed programs. *Distributed Computing*, 1(2), April 1986.
- [Ng86] P. Ng. *Long Atomic Computations*. PhD thesis, Massachusetts Institute of Technology, August 1986.
- [S*85] A. Z. Spector et al. Distributed transactions for reliable systems. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 127–146, ACM, 1985.
- [SS84] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.
- [W*82] R. Williams et al. R*: an overview of the architecture. In *Proceedings of Second International Conference on Databases: Improving Usability and Responsiveness*, ACM, 1982.
- [W*85] M. Weinstein et al. Transactions and synchronization in a distributed operating system. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 115–126, ACM, 1985.
- [Wei84] W. E. Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, 1984.

